The Force: A Highly Portable Parallel Programming Language

by
Harry F. Jordan, Muhammad S. Benten,
Gita Alaghband and Ruediger Jakob

Computer Systems Design Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309-0425

CSDG 89-2
February 1989

The Force: A Highly Portable Parallel Programming Language

Harry F. Jordan, Muhammad S. Benten, Gita Alaghband and Ruediger Jakob

University of Colorado
Department of Electrical & Computer Engineering

Keywords: multiprocessors, shared memory, parallel languages, portability, macro preprocessor

Abstract

Programming multiprocessors is still a highly machine dependent task and parallel programs are rarely portable. In our paper we will explain why the Force parallel programming language has been easily portable between six different shared memory multiprocessors. We show how a two level macro preprocessor allows us to hide low level machine dependencies and to build machine independent high level constructs on top of them. These Force constructs enable us to write portable parallel programs largely independent of the number of processes and independent of the specific shared memory multiprocessor executing them.

1 Introduction

All manufacturers of multiprocessors provide with their machines some support for parallel programming. But this support is very machine dependent and often only at a low level.

In this paper we present a method for providing highly portable language support for parallel programming using macro preprocessors. We show how to construct high level control oriented language structures based on machine independent intermediate level structures, which are in turn based on machine dependent, manufacturer supplied constructs. By doing this, we identify the key machine dependent constructs for multiprocessing support.

The Force [Jor87,JBAR87], a parallel programming language for large scale shared memory multiprocessors has been successfully implemented and ported to a variety of systems using this methodology.

We will begin with an argument for parallel programming and an overview of related work. We briefly present the Force language concepts for parallel programming. In the remainder of the paper a detailed description of the implementation of the Force and its portability to different multiprocessors will be given.

2 Motivation and Related Work

Vectorizing and parallelizing compilers have been used in the past to convert sequential programs into vector and parallel forms. But except in relatively regular cases the compilers have been unable to detect concurrency in sequential programs. More importantly, compilers cannot come up with new, parallel solution algorithms. Therefore parallel programming languages are needed to allow development of parallel programs.

Current multiprocessor manufacturers provide support for parallel programming. But experience shows us that development and maintenance of parallel programs using that support is a complex and difficult task. It requires the scientific or application programmer to have an intrinsic knowledge of the basic parallelism concepts and their use, which is different for each machine.

The tremendous effort spent on design and implementation of programs would only be cost effective if they can be ported to a large class of multiprocessors for hardware independence [LO87]. If the parallel programming language used could be ported to other target machines using their parallel programming tools, then the programs written in that language would also be portable.

The Force has been designed around these ideas. It is a parallel programming language designed for large scale shared memory multiprocessors which evolved in the course of implementing numerical algorithms. Being a language extension to Fortran, it is implemented with a macro preprocessor. Work is not assigned to specific processes, but distributed over the entire force of processes by parallel

constructs. The variables on which work is performed are either uniformly shared among all of the processes or strictly private to a single process. The key ideas embodied in the Force are global parallelism, independence of the number of processes executing a parallel program, high performance of tightly coupled programs, suppression of process management, and reliance on generic synchronizations. "Generic" in this context means that the processes do not have to be explicitly specified for the synchronization operations.

Similar work on multiprocessor programming languages has been done for EPEX/Fortran [DGNP88] and for the Uniform System library on the BBN Butterfly [TC88]. The parallel programming language Linda [GCCC85] and the work at the Oregon Graduate Center on Large Grain Data Flow [Bab84], as well as Dongarra's work on SCHEDULE [DS87] are closely related, because they also address the scheduling problem on multiprocessors.

The implementation of the Force relies on the constructs for process creation, synchronization, and shared memory designation provided by the target multiprocessor. Hiding these machine dependencies from the user by defining a machine independent parallel programming language is one key achievement of the Force.

The Force has been implemented on the HEP, Flex/32, Encore Multimax, Sequent Balance, Alliant FX/8, and Cray-2 multiprocessors.

Implementation of the Force on systems involving several, rather different process models has not been difficult, and porting it between machines with

similar system supported primitives is almost trivial. Given the fairly strong differences between the machines already hosting the Force, we expect no major difficulties in porting the system to any shared memory multiprocessor.

3 Key concepts in the Force programming language

The parallel programming primitives introduced by the Force have been kept conceptually small, each embodying only one concept. The Force language concepts can be divided into four classes: program structuring, variable classification, work distribution and synchronization concepts.

3.1 Parallel program structure

Parallel programs can call subroutines from existing Fortran libraries. In addition, parallel Force subroutines are supported by the Forcesub statement. Such a parallel subroutine is executed by all processes concurrently.

3.2 Variable classification

The Force computational model introduces a new variable classification, which is orthogonal to the Fortran local/common classification. In the same way that local/common specifies name scope between program modules, the shared/private classification of the Force specifies name scope between processes. Shared vari-

ables are shared between the processes executing a Force program, whereas the scope of private variables is restricted to one process.

Data synchronization is supported by the variable class async for asynchronous variables with a full/empty state.

3.3 Work distribution

The distribution of work between the processes can be done at run time, what we call selfscheduled, or at compile time, called prescheduled.

Segments of code that can be executed concurrently, in any order, can be distributed. In case of singly (doubly) nested loops, the loop indices (index pairs) specify concurrently executable sequential streams of code, which are split up in an unspecified way for concurrent execution (DOALL loops). If the concurrently executable code sections are not loop bodies, they can be distributed with the more general Pcase construct. This is a collection of independent sections of code, each executed by a single process. The independent sections can be executed conditionally or unconditionally. Again no specific sequence of execution can be assumed.

The most general concept for concurrent code segments is Askfor [LO83]. This construct provides a means of work distribution in cases where the degree of concurrency is not known at compile time. Rather the program can request during run time that a new concurrent instance of the code segment is executed. A yet unimplemented concept is Resolve, which would partition the set

of processes into subsets executing different parallel code sections.

3.4 Synchronization operations

The Force provides data and control oriented synchronization mechanisms.

Data oriented synchronization is used to synchronize updating of variables which are shared between processes. The shared variable has to be of class Async which associates a full/empty state with the variable's value. The state of the variable is changed with read and write access as an atomic action; to be more specific: A Consume waits for the variable to be full, reads the value and sets it to empty; A Produce waits for the variable to be empty, writes its value and sets it to full. The state can also be tested and initialized to empty.

Control oriented synchronization uses the two well known concepts of critical sections and barriers. At a barrier, all processes wait for each other. One arbitrary process is then allowed to execute the barrier section. All other processes are suspended until the single process leaves the barrier section. Critical sections implement the mutual exclusion condition. Only one process at a given time is allowed to execute within the critical section.

4 The Force Implementation

The implementation of the Force is divided into a set of machine dependent macros and a set of machine independent macros. In what follows we first describe the machine dependencies, the variation between machines, and their ef-

fects on the implementation. Next we describe the machine independent macros which can readily be ported to any shared memory multiprocessor.

4.1 Machine Dependent Force

The following is a list of the machine dependent macros, and these are the only ones we use to implement all higher level language constructs:

- force_environment: declares and initializes the environment variables for the implementation of barriers and selfscheduled loops and a unique process identifier

- define_lock(var): declares a shared lock variable
  init_lock(var): initializes the lock variable to be unlocked
  lock(var): locks the variable
  unlock(var): unlocks the variable

- shared(var): declare a variable as shared among processes
  shared_common(var): declare a COMMON block as shared
  async(var): declare a variable as asynchronous
  async_common(var): declare a COMMON block as asynchronous
  private(var): declare a cariable as private (default)
  private_common(var): declare a COMMON block as private (default)

These macros define a machine independent low level parallel language extension for memory sharing and synchronization operations. Process creation

and termination are also highly machine dependent parts of the Force macro implementation. In the following we will detail the machine dependencies of these constructs.

4.1.1 Process Creation and Termination

The Force has a global parallelism execution model therefore, a Force program is written with the assumption of the existence of a force of processes to execute the program. The processes are created in a Force driver which is generated when the program is preprocessed. The processes are only terminated at the end of the program when a Join statement is executed. The process creation and the necessary synchronization code for joining the created processes are done in the machine dependent driver code.

The following models for process creation have been encountered so far. The standard UNIX fork/join process control model. This model has a large process creation and context switching cost. This prevents fine grained parallelism, unless the parallelism is not enclosed inside the program structure. Encore and Sequent use this model for creation of processes where a complete copy of the data and stack is produced for each forked process. Alliant uses a variation of this model where all data segments are shared and only the stack is considered private. A new copy of the stack is therefore part of the process state.

The second model is that of the HEP multiprocessor. On this machine, one can create processes with a subroutine call. The code of the subroutine is

executed by a new process in parallel with the calling process. A return from the subroutine terminates the subroutine independently of the calling process.

4.1.2 Parallel Environment

A Force programmer must explicitly declare the type and storage class of variables used in his program in the declaration section. In addition, the preprocessor provides a set of variables used to implement the Force constructs for work distribution and synchronization, such as process number, barrier locks and arrival counter, and asynchronous loop index for selfscheduled loops. Because various multiprocessors handle the sharing of memory differently, the macro which processes the user declarations and the parallel environment variables is machine dependent:

On the Flex and HEP multiprocessors, variables are declared shared at compile time. The macro preprocessor simply strips off the word shared and places all shared and asynchronous variables in Fortran COMMON areas, which are shared between the processes. This leads to the simplest implementation, since each separately compilable module declares its shared variables.

On the Sequent, sharing of variables is done at link time. The implementation must provide the linker with the names of all shared variables. The preprocess step is used to create a startup subroutine in the main Force program and in every Force subroutine. The startup routine in each program segment will contain the information about the variables declared for the parallel envi-

ronment in that routine. The startup routine in the main program contains a call statement to the startup routine of every Force subroutine in the program to provide a linkage between all shared variables used throughout the program. To provide this information to the linker, the program is run twice. In the first run, only the startup subroutines will be executed. This run will provide the linker commands to a UNIX shell which will take the commands in a pipe fashion to link and run the complete program with the correct linker information the second time.

On the Encore Multimax, the shared memory is identified at run time. A process similar to the Sequent is used for generating the startup routines and providing the parallel environment. But since the sharing is done at run time, no linker commands have to be generated on this machine, and one run is sufficient. The shared variables are stored in shared pages, and it is in general the programmer's responsibility to ensure that shared variables are within the shared page boundaries and that private variables are not. The Force relieves the programmer from this responsibility by calculating the address of shared pages and padding the extra space at the beginning and the end of the shared area to ensure separation of shared and private declarations. The Alliant implementation is very similar to Encore except that all sharing must start at the beginning of a page.

4.1.3 Lock support

For synchronization operations we can use either the locks provided by the underlying machine, or take advantage of hardware support for produce/consume operations, as was the case on the HEP multiprocessor with its full/empty access state bit. A number of low level macros are used as generic lock mechanisms for initializing, locking, and unlocking asynchronous variables. Only these low level macros are affected when porting the Force to other machines. The various types of locks provided by different systems are:

- software locks: spinning with test&set on shared variables (Sequent, Encore)
- system call locks: operating system handles a list of locked processes in cooperation with the scheduler (Cray)
- combined lock: spinlock for limited time, then make operating system call (Flex)

In some machines, locks may be scarce resources. On these machines, some parallel programs may not execute as efficiently as others if a large number of asynchronous variables are needed.

4.2 Machine Independent Force

The Force implementation has been easily portable. With the exception of the macros just described, all macros are machine independent. This independence

of the specific multiprocessor was achieved by the two layer implementation: The lower level of the Force macros hides the manufacturer provided multiprocessing extensions under machine independent, low level parallelism primitives for memory sharing and locks. Based on these few low level constructs, it is possible to construct the higher level parallel programming concepts completely machine independent.

The identification of the basic parallelism constructs is the critical point. On one side, these basic macros should be as efficient as possible for most machines. On the other side, they have to be general enough to construct all needed high level constructs. Our machine independent macros can be divided into utility macros, statement macros, and internal macros:

The utility macros are used for processing the text of the program and facilitating its conversion to the target form. Examples are be macros for returning the first element of a list, storing and retrieving definitions, concatenating and truncating arguments, and deletion of dimensions for common declarations.

The statement macros explicitly process the Force language constructs in programs. They translate them into Fortran code and low level machine dependent macro calls. Some examples will show how this is done:

Force: This macro defines the main Force program. It calls the machine dependent macros for parallel Force environments and startup subroutine generation. It will also set a flag indicating that a main program has been defined. All subsequently encountered Force subroutines will provide information about

their shared variables to the main program by inserting a call statement to their startup routine in the main program's startup routine.

Externf: Since the Fortran loader has no prior knowledge of the Force environment, external Force subroutine declarations are needed when Force subroutines are to be separately compiled. The Externf macro generates the startup subroutine calls in the main program.

Barrier: This macro uses generic lock macros to implement the entry code for a barrier construct using the Force parallel environment variables for barrier locks and arrival reporting. [AJ87]

DOALL: There are two variations of DOALLs for work distribution in the Force: prescheduled and selfscheduled DO loops. The prescheduled DO loop is completely machine independent, since only the number of executing processes is needed to distribute the index values among processes. The selfscheduled DO loop is more complex and requires a shared variable as the loop index which must be updated by processes looking for more work. Therefore this macro will call generic machine dependent macros for the declaration of shared variables and for synchronization. But the macro itself is not changed in the port process.

As an example, we will show the macro expansion for the following selfscheduled simple loop:

```
        Selfsched DO 100 K = START, LAST, INCR
        (* LOOPBODY *)
100     End Selfsched DO
```

This will be replaced by:

```
C loop entry code
        lock(BARWIN)
        IF (ZZNBAR .EQ. 0) THEN
C initialize loop index
           K_shared = START
        END IF
C report arrival of processes
        ZZNBAR = ZZNBAR + 1
        IF (ZZNBAR .EQ. number_of_processes) THEN
           unlock(BARWOT)
        ELSE
           unlock(BARWIN)
        END IF
C selfscheduled loop index distribution
100     lock(LOOP100)
C get next index value
        K = K_shared
        K_shared = K + INCR
        unlock(LOOP100)
C test for completion
        IF ((INCR .GT. 0 .AND. K .LE. LAST) .OR.
              (INCR .LT. 0 .AND. K .GE. LAST)) THEN

           (* LOOPBODY *)

           GO TO 100
        ENDIF
C loop exit code
        lock(BARWOT)
C report exit of processes
        ZZNBAR = ZZNBAR - 1
        IF (ZZNBAR .EQ. 0) THEN
           unlock(BARWIN)
        ELSE
           unlock(BARWOT)
        END IF

(K_shared and LOOP100 are the declared shared INTEGER and
LOGICAL variables; BARWIN, BARWOT and ZZNBAR ensure that all
processes have left the loop, before it can be reentered)
```

Pcase: Pcase is a similar construct to DOALL, which distributes different

single stream code blocks over the processes of the Force. Each block may be associated with a condition, and any number of conditions may be true simultaneously. The prescheduled version of this macro allocates the blocks sequentially to the processes and is thus completely machine independent. A selfscheduled Pcase is similar to the selfscheduled do

loop in that an asynchronous variable is needed for work distribution.

Produce/Consume: Force uses the full/empty state for asynchronous variables. With the exception of the HEP computer which provided a hardware full/empty state for every memory cell, all other machines require the use of two locks for implementation of the full/empty state. In this scheme, an asynchronous variable must be empty before a process can produce it. Two locks E and F are associated with each asynchronous variable. An empty state corresponds to E being locked and F unlocked. A full state corresponds to F being locked and E unlocked.

A Produce process goes through the following steps to produce a value for an empty asynchronous variable and to leave its state as full:

Lock F

Write to the asynchronous variable

Unlock E

Other processes trying to produce the same variable will find F locked and will wait.

A Consume process executes the following steps to read the value of a full

asynchronous variable and to leave its state empty:

Lock E

Read from the asynchronous variable

Unlock F

While a Produce is in progress, a Consume process will wait until E is unlocked. The lock and unlock operations are simply calls to the low level macros corresponding to the generic lock mechanisms mentioned above. This simply allows the logical steps for Produce/Consume synchronization primitives to be carried out correctly once the low level lock mechanism of each machine is provided.

Void: This macro is used to set the state of an asynchronous variable to empty regardless of its previous state. It is mainly used to initialize the state of asynchronous variables and uses a similar procedure as above.

Internal macros: Finally, internal macros are used by the statement macros to insert synchronization codes at various locations in the body of some Force constructs. In the above example of a selfscheduled DO loop a macro to synchronize the processes entering the loop, a macro to test if processes must repeat executing the loop body, and a macro to test completion of the work by all processes are used. These intermediate macros will call the generic machine dependent macros.

4.3 Implementation Structure

The Force has been easily portable, because only a small portion of the preprocessor is machine dependent and only a rudimentary set of parallelism support is needed from a machine.

In a UNIX environment, the compilation of Force programs proceeds in three steps:

The stream editor sed translates the Force syntax into parameterized function macros. Then the macro processor m4 replaces the function macros with Fortran code and the language extensions supporting parallel programming. This replacement occurs in two steps, as described above. The machine dependent driver module is put at the beginning of the code. Finally the manufacturer provided Fortran compiler and linker process the macro expanded code with the appropriate option settings.

5 Conclusions

The Force is one of the few parallel programming languages that have been implemented on a wide variety of multiprocessors. It allows users and researchers to move programs between different machines, a capability crucial to further development in parallel programming. We have presented how macro preprocessors enabled us to implement the Force on various machines without having to make many changes to the preprocessor. By constructing a low level parallel

language we were able to built high level, control oriented structures largely independent of the specific, machine dependent multiprocessing extensions.

References

[AJ87] N.S. Arenstorf and H.F. Jordan. *Comparing Barrier Algorithms.* ECE Tech. Rept. 87-1-2, Computer System Design Group, Electrical & Computer Engineering Dept., Univ. of Colorado, Boulder, Jun. 1987.

[Bab84] R.G. Babb. Parallel processing with Large Grain Data Flow Techniques. *IEEE Computer*, 17:55–61, Jul. 1984.

[DGNP88] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. A single-program-multiple-data computational model for EPEX/Fortran. *Parallel Computing*, 7:11–24, 1988.

[DS87] J.J. Dongarra and D.C. Sorensen. SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs. In *The Characteristics of Parallel Algortithms*, pages 363–394, MIT Press, 1987.

[GCCC85] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel Programming in Linda. In *Proc. 1985 Intl. Conf. Parallel Processing*, pages 255–263, 1985.

[JBAR87] H.F. Jordan, M.S. Benten, N.S. Arenstorf, and A.V. Ramanan. *Force User's Manual.* ECE Tech. Rept. 86-1-4R, Computer System Design Group, Electrical & Computer Engineering Dept., Univ. of Colorado, Boulder, Oct. 1987.

[Jor87] H.F. Jordan. The Force. In *The Characteristics of Parallel Algorithms*, pages 395–436, MIT Press, 1987.

[LO83] E.L. Lusk and R.A. Overbeek. *Implementation of monitors with macros: A programming aid for the HEP and other parallel processors.* Technical Report ANL-83-97, Argonne National Lab, IL, Dec. 1983.

[LO87] E.L. Lusk and R.A. Overbeek. A minimalist approach to portable, parallel programming. In *The Characteristics of Parallel Algorithms*, pages 351–362, MIT Press, 1987.

[TC88] R.H. Thomas and W. Crowther. The Uniform System: An approach to runtime support for large scale shared memory parallel processors. In *Proc. 1988 Int. Conf. Parallel Processing*, pages 245–254, 1988.

4. Title and Subtitle
The Force: A Highly Portable Parallel Programming Language

5. Report Date
February 1989

6.

7. Author(s) Harry F. Jordan, Muhammad S. Benten, Gita Alaghband and Ruediger Jakob

8. Performing Organization Rept. No.

9. Performing Organization Name and Address
Computer Systems Design Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309-0425

10. Project/Task/Work Unit No.

11. Contract/Grant No.
N00014-86-k-0204

12. Sponsoring Organization Name and Address
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217-5000

13. Type of Report & Period Covered
Interim

14.

15. Supplementary Notes
Also supported in part by NASA Langley Research Center under grant NAG-1-640.

16. Abstracts

Programming multiprocessors is still a highly machine dependent task and parallel programs are rarely portable. In our paper we will explain why the Force parallel programming language has been easily portable between six different shared memory multiprocessors. We show how a two level macro preprocessor allows us to hide low level machine dependencies and to build machine independent high level constructs on top of them. These Force constructs enable us to write portable parallel programs largely independent of the number of processes and independent of the specific shared memory multiprocessors executing them.

17. Key Words and Document Analysis. 17a. Descriptors

multiprocessors
shared memory
parallel languages
portability
macro preprocessor

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement

19. Security Class (This Report) UNCLASSIFIED

21. No. of Pages 19

20. Security Class (This Page) UNCLASSIFIED

22. Price